FRPA: A Framework for Recursive Parallel Algorithms

David Eliahu
Omer Spillinger
Armando Fox
James Demmel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-28
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-28.html

May 1, 2015

FRPA: A Framework for Recursive Parallel Algorithms

David Eliahu*†, Omer Spillinger§†, Armando Fox‡†, and James Demmel¶

***Abstract*—Recursion continues to play an important role in high-performance computing. However, parallelizing recursive algorithms while achieving high performance is nontrivial and can result in complex, hard-to-maintain code. In particular, assigning processors to subproblems is complicated by recent observations that communication costs often dominate computation costs. Previous work [1]–[3] demonstrates that carefully choosing which divide-and-conquer steps to execute in parallel (breadth-first steps) and which to execute sequentially (depth-first steps) can result in significant performance gains over naïve scheduling. Our Framework for Recursive Parallel Algorithms (FRPA) allows for the separation of an algorithm's implementation from its parallelization. The programmer must simply define how to split a problem, solve the base case, and merge solved subproblems; FRPA handles parallelizing the code and tuning the recursive parallelization strategy, enabling algorithms to achieve high performance. To demonstrate FRPA's performance capabilities, we present a detailed analysis of two algorithms: Strassen-Winograd [1] and Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication (CARMA) [3]. Our single-precision CARMA implementation is fewer than 80 lines of code and achieves a speedup of up to 11× over Intel's Math Kernel Library (MKL) [4] matrix multiplication routine on "skinny" matrices. Our double-precision Strassen-Winograd implementation, at just 150 lines of code, is up to 45% faster than MKL for large square matrix multiplications. To show FRPA's generality and simplicity, we implement six additional algorithms: mergesort, quicksort, TRSM, SYRK, Cholesky decomposition, and Delaunay triangulation [5]. FRPA is implemented in C++, runs in shared-memory environments, uses Intel's Cilk Plus [6] for task-based parallelism, and leverages OpenTuner [7] to tune the parallelization strategy.**

I. Introduction

Recursion continues to be relevant in many high-performance computing (HPC) algorithm implementations. Although the recursive definitions of these algorithms may be quite simple, parallelizing them is nontrivial [2], [3]. Algorithm developers must make decisions about load balancing, tuning, and parallelization schemes. With every optimization that is made to improve parallel performance, the underlying algorithm is obscured. Simple algorithms that can be coded sequentially in tens of lines of code may require thousands of lines for their highly-optimized parallel counterparts [8].

Recent research in communication-avoiding algorithms has lead to the BFS/DFS parallelization strategy [2]. Breadth-first steps (BFS steps) and depth-first steps (DFS steps) are alternate ways to allocate processors to solve subproblems in a parallel recursive algorithm. At a BFS step, all of the subproblems are solved in parallel; at a DFS step, the subproblems are solved sequentially. In general, BFS steps reduce communication costs and expose parallelism, but require extra memory relative to DFS steps [1]–[3]. BFS steps expose parallelism by creating tasks (i.e. the subproblems) that can be solved in parallel, and decrease future communication because fewer processors will be assigned to each subproblem. However, extra memory is required to execute a BFS step because as the subproblems are solved in parallel, each one concurrently allocates the memory it needs. In a DFS step, subproblems are solved one at a time, reducing concurrent memory usage.

Because of their recursive structure, BFS/DFS algorithms are cache-, processor-, and network-oblivious [9]–[11]. An optimal interleaving of BFS and DFS steps that remains within the available memory gives a communication-optimal algorithm both for classical matrix multiplication [3] and Strassen-Winograd's algorithm [1], [2] for any memory size. In a shared-memory environment, the BFS/DFS interleaving affects the algorithm's memory footprint, cache access pattern, the number of simultaneous threads of execution, and the size of the base cases. All of these effects result in machine-specific performance variation for different BFS/DFS interleavings.

Our Framework for Recursive Parallel Algorithms (FRPA) aims to separate the algorithm's kernel from its parallelization and optimization. By abstracting away the parallelism and providing a simple API, FRPA allows programmers to focus solely on expressing their algorithm's recursive structure. FRPA handles the rest: parallelizing the code, autotuning the BFS/DFS interleaving, and autotuning algorithm-specific parameters. FRPA also provides programmers automatic access to a custom memory tracking tool, which can be used to evaluate a program's memory footprint and identify memory leaks.

FRPA's goal is to make implementing parallel recursive algorithms as easy as implementing their sequential coun-

*deliahu@eecs.berkeley.edu
†EECS Department, UC Berkeley, Berkeley, CA 94720
§omers88@eecs.berkeley.edu
‡fox@cs.berkeley.edu
¶demmel@cs.berkeley.edu, CS Division and Mathematics Department, UC Berkeley, Berkeley, CA 94720

terparts. To demonstrate the API's simplicity and generality, we implement eight algorithms: Strassen-Winograd [1] and CARMA [3] matrix multiplication, mergesort and quicksort sorting algorithms, solving triangular systems of linear equations with many right-hand sides (TRSM), symmetric rank-k matrix-matrix updates (SYRK), decomposition of a symmetric positive-definite matrix into the product of a lower triangular matrix and its transpose (Cholesky decomposition), and a triangulation algorithm that maximizes the angles of the triangles (Delaunay triangulation [5]).

Of course, FRPA can only be useful within the HPC community if it also delivers high performance. To highlight FRPA's efficiency, we perform an in-depth analysis of two of the aforementioned algorithms: Strassen-Winograd and CARMA. We show that CARMA's implementation in FRPA consistently outperforms the original CARMA implementation, achieving up to a 57% speedup. The FRPA implementation also *significantly* outperforms Intel's MKL [4] (version 10.3.9) matrix multiplication routine for "skinny" matrix dimensions (up to $11\times$ in speedup in single precision). We also demonstrate that our implementation of Strassen-Winograd matrix multiplication achieves high-performance: it outperforms MKL by up to 45% on large square matrices (double precision), and exceeds the theoretical peak performance of any classical matrix multiplication algorithm by up to 26%.

By implementing our framework in C++, we are able to design an easy-to-use object-oriented API without sacrificing performance. FRPA uses Intel's Cilk Plus [6] for task-level parallelism, and Cilk Plus handles task scheduling and load balancing. Additionally, FRPA uses OpenTuner [7] to tune the BFS/DFS interleaving and all algorithm-specific tuning parameters. FRPA performs best when the platform supports many parallel threads of execution.

A. Contributions

- We design and implement FRPA, a simple framework for expressing recursive parallel algorithms on shared-memory platforms. Programmers define the recursive structure of an algorithm, and the framework automatically handles parallelization and autotuning, thereby providing users without extensive computer science expertise a tool for generating high-performance code.
- We demonstrate that the FRPA implementations of CARMA and Strassen-Winograd exhibit very high performance: both significantly outperform the Intel MKL [4] dense matrix multiplication routine.
- We implement a total of eight algorithms using FRPA to illustrate its generality and simplicity. These algorithms, which include matrix multiplication, mergesort, Delaunay triangulation, and Cholesky decomposition, vary widely in their recursive pattern and complexity.
- We use OpenTuner [7] to autotune the BFS/DFS parallelization strategy of algorithms in FRPA, and we expose OpenTuner's API to allow developers to tune algorithm-specific parameters. We study the effect of tuning the BFS/DFS interleaving for Strassen-Winograd and CARMA.

B. Paper Organization

We begin by surveying related work in Section II. We describe the API in Section III and define its syntax in Section IV. In Sections V and VI, we discuss the implementation of FRPA itself. We then present detailed evaluations of two of the algorithms we implemented (Strassen-Winograd in Section VII and CARMA in section VIII). We analyze the performance patterns of these algorithms and OpenTuner's convergence in Sections IX and X, respectively. Finally, we discuss opportunities for future work in Section XI. We include brief discussions of the other six algorithms we implemented in the Appendix.

II. Related Work

Making parallelism accessible to people without substantial computer science expertise is a well-known challenge. Early attempts to accomplish this were language based. A Nested Data-Parallel Language (NESL) [8] was designed with four goals in mind: to support parallelism via data-parallel constructs based on sequences, to support complete nested parallelism, to generate efficient code for a variety of architectures, and to be suitable for describing parallel algorithms. We acknowledge the value of designing a programming language such as NESL from the ground up for its ability to be optimized for teaching and prototyping parallel algorithms. However, we believe that this approach has an inherent obstacle to widespread use: the learning curve is high for users who are only familiar with mainstream programming languages such as C, C++, or Python. By providing a simple interface to the complex and powerful parallelism capabilities of an existing programming language, FRPA achieves many of NESL's goals without forcing users to adopt an entirely new language.

The researchers behind the Cell superscalar (CellSs) [12] and SMP superscalar (SMPSs) [13] programming environments also address the challenge of providing an easy-to-use, flexible, and portable programming model for high-performance parallel programming on SMP and multiprocessor architectures. CellSs and SMPSs require pragmas to identify tasks (atomic routines that operate over a set of parameters). The compiler and runtime library detect task calls as well as their interdependencies and parallelize them. Although FRPA is less general than SMPSs, its focus on recursive algorithms allows it to provide more guidance for programmers as well as advanced optimizations such as autotuning the BFS/DFS interleaving.

PATUS [14] is a code generation and autotuning framework that focuses on stencil computations. Although it doesn't support recursive algorithms, its overall concept is similar to FRPA's. Users define the algorithm's kernel using C-like syntax, which the framework parallelizes and optimizes for the hardware platform using domain-specific knowledge that is not included in general-purpose compilers.

Another relevant area of HPC research is heterogeneous architectures. XKaapi [15], a runtime system for data-flow task programming on heterogeneous architectures, supports multi-CPU and multi-GPU architectures. This framework consists of a locality-aware work stealing algorithm, a fully asynchronous task execution strategy on GPUs, low overhead tasks, and lazy computation of dependencies. These optimizations allow algorithms such as Cholesky decomposition to achieve high performance on systems with multiple CPUs and GPUs.

One of the most recent parallel code generation frameworks, Huckleberry [16], has goals that are similar to FRPA's. Huckleberry accepts sequential recursive divide-and-conquer programs as input, and outputs parallel implementations for multiprocessor machines. However, FRPA is a much simpler framework due to its modular design. In FRPA, task parallelism is entirely handled by Cilk Plus. In addition, communication-avoidance is described by BFS/DFS interleavings and optimized by OpenTuner. By leveraging existing tools such as Cilk Plus and OpenTuner, FRPA's API remains simple without sacrificing performance.

Finally, PetaBricks [17] is an implicitly parallel language and compiler that automates algorithm selection at different depths of a recursive algorithm. PetaBricks provides a framework for specifying multiple ways to solve a problem, and a built-in autotuner selects which algorithm to run based on input size. FRPA is similar to PetaBricks in that it uses task-based parallelism and a work-stealing scheduler. However, unlike PetaBricks, FRPA allows for direct tuning of the BFS/DFS parallelization strategy and supports algorithms with inputs of arbitrary dimensions. Also, FRPA uses OpenTuner to handle autotuning. OpenTuner, which was created by many of the same researchers who previously developed PetaBricks, uses a different tuning strategy that generally outperforms the built-in PetaBricks autotuner [7].

The BFS/DFS approach that FRPA uses was first applied to distributed-memory architectures in the implementation of "Communication-Optimal Parallel Strassen" [1]. The technique was later extended to shared-memory machines in "Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication" (CARMA) [3]. In this paper, we generalize the work from CARMA and extend the BFS/DFS approach to arbitrary recursive algorithms in shared memory.

III. API Description

To implement a recursive problem in FRPA, a developer must define how the problem splits into subproblems, how to solve the base case, and how subproblems are merged once solved. These are the only required functions of a recursive algorithm in FRPA, but there are additional features and tools that can improve an algorithm's performance. In this section, we describe FRPA's API at a high level (see Section IV for syntax).

A. Basic API

To implement a parallel recursive algorithm in FRPA, one must define how to split a problem into subproblems (`split()`), how to solve the base case (`runBaseCase()`), and how to merge a collection of solved subproblems (`merge()`). Once an algorithm is properly defined, it can be instantiated and solved using FRPA. FRPA requires that the BFS/DFS interleaving of the parallel recursion be specified at runtime. See Section IV-B for syntax and additional details.

B. Sequential `split()` and `merge()`

As described in Section I, FRPA may or may not choose to execute subproblems in parallel. In the event that FRPA executes the subproblems sequentially (known as a depth-first step, or DFS), there may be an opportunity to optimize the `split()` and `merge()` functions.

To allow optimizations that can only be done in the case of a DFS step, FRPA exposes two optional methods: `splitSequential()` and `mergeSequential()`. These routines are called in place of their non-sequential counterparts during a DFS step. See Section IV-C for the syntax of `splitSequential()` and `mergeSequential()`.

One example of when `splitSequential()` and `mergeSequential()` are useful is in CARMA. CARMA is a matrix multiplication algorithm that recursively splits the longest of the three dimensions (i.e. m, k, or n, where an $m \times k$ matrix is multiplying a $k \times n$ matrix). In the event that the middle dimension k is split, CARMA must avoid race conditions by allocating a temporary matrix to hold the results of the second subproblem. However, if both subproblems will be executed sequentially, there are no race conditions and there is no need to allocate the temporary matrix. In our CARMA implementation, `splitSequential()` is the same as `split()` but without the extra memory allocation. Also, because both CARMA subproblems write to the same matrix in the DFS case, `mergeSequential()` need not merge the temporary matrix with the result matrix. See Section VIII for an explanation of the algorithm, and the first example in Figure 5 for the case where the temporary matrix must be allocated when running in parallel.

C. Expressing Base Case Constraints

Certain types of recursive problems have constraints on whether the base case can or must be called. To specify these constraints, the developer may implement `canRunBaseCase()` and `mustRunBaseCase()` (see Section IV-D for syntax).

`canRunBaseCase()` is used to specify whether or not a particular instance of the problem *can* be solved with the base case. We use `canRunBaseCase()` in our implementation of Delaunay Triangulation (see Appendix F). In the algorithm, the problem can only be solved with the base case if the number of points is less than three. Otherwise, the base case cannot be called and the algorithm must recurse further.

`mustRunBaseCase()` is used to specify whether or not an instance of the problem *must* be solved with the base case. We use `mustRunBaseCase()` in our implementation of Quicksort, among others (see Appendix B). In Quicksort, the base case must be called if the length of the array is less than or equal to one.

D. Memory Tracking

FRPA provides a built-in tool for tracking the memory usage of an algorithm. By default, FRPA will print three values at the completion of the program: `current_memory`, `max_memory`, and `total_memory`. `current_memory` is the amount of memory that has been allocated and not freed; this value should always be zero at a program's completion (otherwise there are memory leaks). `max_memory` is the *maximum* amount of memory allocated and not yet freed at any time during the execution of the program. In other words, `max_memory` is the highest value that `current_memory` ever reached. Finally, `total_memory` is the total amount of memory allocated during the life of the program. All three values are recorded in bytes.

In addition to reporting these three values upon the completion of a program, FRPA enables the developer to query these values at any point during a program's execution. See Section IV-E for the full API.

FRPA's memory tracker can analyze the memory footprint of any algorithm implemented in FRPA. This may be used to guide more efficient algorithms, and was used in this manner to minimize the number of memory allocations performed in our Strassen-Winograd implementation. Another use case is to ensure that an implementation contains no memory leaks (i.e. `current_memory` is zero at the end of the program's execution).

E. Parallelizing `split()` *and* `merge()`

To take full advantage of a machine's parallelism, it may be necessary to parallelize the `split()` and/or `merge()` functions. Significant computation may be performed within `split()` and/or `merge()`. Before many BFS steps have been executed, there are idle processors that may be put to good use in these routines.

We parallelized the `split()` and `merge()` functions in our implementation of Strassen-Winograd. We found that for this particular algorithm, because the majority of the computation occurs in the base case multiplications, parallelizing `split()` and `merge()` did not have a significant effect on performance. We therefore removed this optimization in the final version of our code.

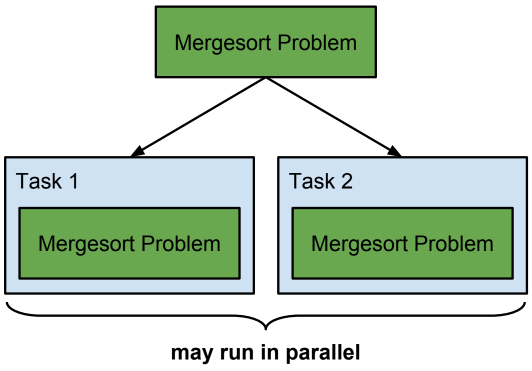


Fig. 1: Illustration of how a `MergesortProblem` splits into subproblems.

IV. API Syntax

In order to implement a recursive algorithm in FRPA, developers must adhere to FRPA's API. In this section, we provide the syntax for this API.

A. Custom Datatypes

FRPA uses two custom datatypes to help the programmer define recursive problems.

`Problem`:
: Defined in `Problem.h` and `Problem.cpp`. Every recursive algorithm implemented in FRPA must be a subclass of type `Problem`.

`Task`:
: Defined in `Task.h` and `Task.cpp`. Encodes how a `Problem` is split into subproblems (subproblems are also of type `Problem`). Every `Problem` splits into a C++ vector of `Task`s. A `Task` is essentially a list of `Problem`s. All `Task`s may be executed in parallel, so there can be no dependencies between `Problem`s in separate `Task`s. However, FRPA enforces that all `Problem`s within a `Task` are run sequentially. Therefore, every `Problem` within a `Task` may depend on all previous `Problem`s within that same `Task`.

To illustrate the distinction between `Task`s and `Problem`s, consider a simple example: two-way mergesort. A `MergesortProblem` splits into two subproblems, also of type `MergesortProblem`. Because these subproblems are independent, a `MergesortProblem` splits into two `Task`s (one for each subproblem). See Figure 1 for an illustration of mergesort.

Now consider a more complicated example: Triangular Matrix System Solver (TRSM). A `TrsmProblem` splits into two sets of subproblems (see Appendix C for an explanation of the algorithm). The sets of subproblems are independent, but all `Problem`s within each set must be executed in sequence. In FRPA, each set maps to a `Task`, and each `Task` contains the `Problem`s associated with that set. See Figure 2 for an illustration of TRSM.

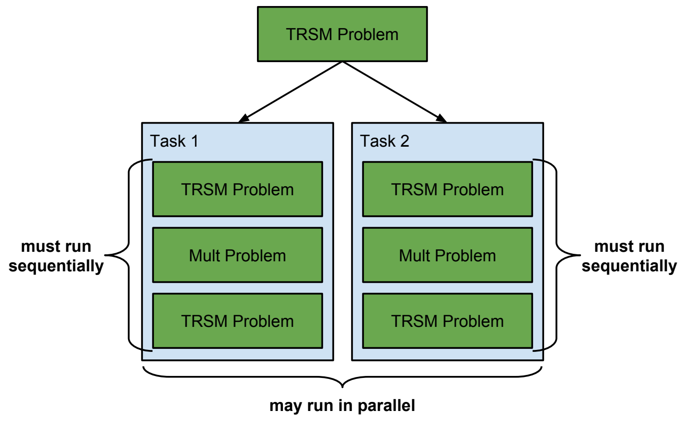


Fig. 2: Illustration of how a `TrsmProblem` splits into subproblems.

Note that a `TrsmProblem` splits into instances of not only `TrsmProblem`, but also of `MultProblem`. This is an example of algorithm composition. FRPA supports algorithm composition in that `Problems` may split into instances of any other type of `Problem`. SYRK and Cholesky are other `Problems` that are composed of heterogeneous subproblems (see Appendix D and Appendix E for more details).

B. Basic API

As described in Section III-A, the developer must specify how to split a problem into subproblems, how to solve the base case, and how to merge a collection of solved subproblems. Developers provide this information by subclassing `Problem` and implementing three methods:

`std::vector<Task*> split()`
: This method defines how a `Problem` is split into subproblems. `split()` returns a vector of `Task` objects. Recall that a `Task` object is essentially an ordered list of `Problems`; `Tasks` are independent, but `Problems` within a `Task` may share dependencies (see Section IV-A for more details).

`void merge(std::vector<Problem*> subproblems)`
: This function specifies how to merge a vector of solved subproblems.

`void runBaseCase()`
: This method defines how to solve an instance of the `Problem`.

In addition to implementing these three functions, the developer must store state with each instance of the `Problem`. `split()` initializes this state for the subproblems, `runBaseCase()` operates on this state, and `merge()` merges this state back into the original `Problem`. The state associated with a `MergesortProblem`, for example, is the array to be sorted and the length of that array.

The framework has only one entry point: `solve()`. `solve()` requires two arguments:

`Problem* problem`
: An instance of type `Problem` that is to be solved.

`std::string interleaving`
: A string representing the BFS/DFS interleaving of the parallel recursion. `interleaving` is a permutation of 'B' and 'D' characters. For example, a valid `interleaving` is 'BBDB.'

To illustrate the basic API, we provide the actual code for mergesort. See Listing 1 for `MergesortProblem.h`, Listing 2 for `mergesort_harness.cpp`, and Listing 3 for `MergesortProblem.cpp`.

C. *Sequential* `split()` *and* `merge()`

To allow optimizations that can only be done in the case of a DFS step (see Section III-B for an example), FRPA exposes two optional methods:

`std::vector<Problem*> splitSequential()`
: This function is called in place of `split()` during a DFS step. Note that `splitSequential()` returns a vector of `Problems`, unlike `split()` which returns a vector of `Tasks`. This is because all subproblems will be executed sequentially, so there is no need to specify which subproblems may run in parallel.

`void mergeSequential`
`(std::vector<Problem*> subproblems)`
: This function is called in place of `merge()` during a DFS step.

If `splitSequential()` and `mergeSequential()` are not implemented by the algorithm, FRPA falls back to the algorithm's default implementation of `split()` and `merge()`.

D. *Expressing Base Case Constraints*

To specify constraints on whether the base case can or must be called, an algorithm may override the following methods:

`bool canRunBaseCase()`
: Specifies whether or not the problem *can* be solved with the base case. If `canRunBaseCase()` is not implemented by the `Problem`, it defaults to `true` (i.e. the base case can always be executed). This default implementation is desired for algorithms like CARMA or Strassen-Winograd, where the base case is the MKL matrix multiplication routine (which can be called at any point during the recursion).

`bool mustRunBaseCase()`
: This method is used to specify whether or not the problem *must* be solved with the base case. If `mustRunBaseCase()` is not implemented by the `Problem`, it defaults to `false` (i.e. a problem can always be split for deeper recursion).

See Section III-C for a description and examples of when `canRunBaseCase()` and `mustRunBaseCase()` may be used.

```cpp
#include "Task.h"
#include "Problem.h"

class MergesortProblem: public Problem {

private:
  double* A;
  int length;

public:
  MergesortProblem(double* A, int length);
  bool mustRunBaseCase();
  void runBaseCase();
  std::vector<Task*> split();
  void merge(std::vector<Problem*> subproblems);
};
```

Listing 1: MergesortProblem.h

```cpp
#include "harness.h"
#include "framework.h"
#include "MergesortProblem.h"

double* randomArray(int length) {
  double* A = (double*) malloc(length * sizeof(
      double));;
  for (int i = 0; i < length; i++) {
    A[i] = 10000 * drand48() - 1;
  }
  return A;
}

int main(int argc, char **argv) {
  srand48(time(NULL));

  int length = atoi(argv[1]);
  std::string interleaving = (argc > 2) ? argv[2] :
      "";

  double* A = randomArray(length);
  MergesortProblem* problem = new MergesortProblem(A
      , length);

  struct timeval start, end;
  gettimeofday(&start, NULL);

  Framework::solve(problem, interleaving);

  gettimeofday(&end, NULL);
  double mergesortTime = (end.tv_sec - start.tv_sec)
       + 1.0e-6 * (end.tv_usec - start.tv_usec);

  printf("length: %d, interleaving: %s, time: %f
      seconds\n", length, interleaving.c_str(),
      mergesortTime);

  free(A);
  delete problem;

  return 0;
}
```

Listing 2: mergesort_harness.cpp

```cpp
#include "MergesortProblem.h"
#include "memorytracking.h"

int cmp(const void* x, const void* y) {
  double xx = *(double*) x;
  double yy = *(double*) y;

  if (xx < yy) return -1;
  if (xx > yy) return  1;
  return 0;
}

MergesortProblem::MergesortProblem(double* A, int
    length) {
  this->A = A;
  this->length = length;
}

bool MergesortProblem::mustRunBaseCase() {
  return (length <= 1);
}

void MergesortProblem::runBaseCase() {
  qsort(A, length, sizeof(double), cmp);
}

std::vector<Task*> MergesortProblem::split() {
  int midpoint = length/2;
  std::vector<Task*> tasks (2);
  tasks[0] = new Task(new MergesortProblem(A,
      midpoint));
  tasks[1] = new Task(new MergesortProblem(A +
      midpoint, midpoint + (length % 2)));
  return tasks;
}

void MergesortProblem::merge(std::vector<Problem*>
    subproblems) {
  MergesortProblem* s1 = (MergesortProblem*)
      subproblems[0];
  MergesortProblem* s2 = (MergesortProblem*)
      subproblems[1];

  double* merged = (double*) malloc(length * sizeof(
      double));
  int p1 = 0, p2 = 0;
  int i = 0;

  // merge
  while (p1 != s1->length && p2 != s2->length) {
    if (s1->A[p1] < s2->A[p2]) {
      merged[i] = s1->A[p1];
      p1++;
    } else {
      merged[i] = s2->A[p2];
      p2++;
    }
    i++;
  }

  // move leftovers
  if (p2 == s2->length) {
    memcpy(A+i, s1->A+p1, (length-i) * sizeof(double
        ));
  }

  // copy merged array into A
  memcpy(A, merged, i * sizeof(double));

  free(merged);
}
```

Listing 3: MergesortProblem.cpp

E. Memory Tracking

As described in Section III-D, FRPA provides a built-in tool for tracking the memory usage of an algorithm. The memory tracker only tracks memory allocated within the framework (i.e. memory allocated by a `Problem` in `split()`, `merge()`, or `runBaseCase()`). In order to enable FRPA's memory tracker, an algorithm must include the `memorytracking.h` header file and must be compiled with the `-DDEBUG` flag. Default memory logging can be disabled by compiling with the `-DTERSE` flag.

In addition to reporting `current_memory`, `max_memory`, and `total_memory` at the completion of a program (see Section III-D for definitions), FRPA provides the following functions to query these values at any point during a program's execution:

`long Memory::getCurrent()`
: Returns the current value of `current_memory`.

`long Memory::getMax()`
: Returns the current value of `max_memory`.

`long Memory::getTotal()`
: Returns the current value of `total_memory`.

`void Memory::reset()`
: Reset all memory counters to zero.

Memory tracking in FRPA is implemented using only atomic operations (namely atomic additions and `compare_and_swap`) for minimal impact on performance.

F. Parallelizing `split()` and `merge()`

As Section III-E explains, it may be necessary to parallelize the `split()` and/or `merge()` functions to take full advantage of a machine's parallelism. To allow for this optimization, FRPA tracks the number of BFS steps that have been taken. This value can be queried in the `split()` and `merge()` functions as `Problem::numBs`. When desired, `numBs` can be used to determine whether to parallelize `split()` and `merge()` and how much parallelism to exploit.

FRPA only exposes the number of BFS steps taken because the number of idle processors depends only on this value. For example, CARMA splits into two subproblems. When a BFS step is executed, the number of idle processors is cut in half (assuming there still are idle processors). Therefore, after three BFS steps there are up to eight processors in use, and the rest are idle. No additional processors are utilized when a DFS step is executed, so the number of DFS steps has no effect on the number of available processors.

We utilized this technique to parallelize the `split()` and `merge()` functions of Strassen-Winograd. We used the OpenMP `parallel for` loop, and set the number of OpenMP threads based on the value of `numBs`. As described in Section III-E, this optimization did not result in significant performance improvements and was removed in the final version of our code.

V. Framework Implementation

A. Language Choice

We chose to implement FRPA in C++. C++ allows us to use an object-oriented design without sacrificing performance. Inheritance and virtual functions make our implementation of `Problems` elegant and simple. Moreover, many high-performance scientific computing libraries are accessible from C++ and can easily be called in the base case of algorithms written in FRPA.

B. Parallelization

FRPA's parallelism is handled by the Cilk Plus runtime system. Cilk multi-threaded computations have a directed acyclic graph structure. This DAG is constructed and scheduled dynamically. Each vertex of the DAG represents a thread that, once invoked, can run to completion without blocking. Cilk threads can `spawn` child threads that may run concurrently with their parent. If a thread spawns child threads, it must also spawn a *successor* thread to receive the children's return values. Edges in the graph connect parent threads with child threads, and connect threads that have data dependencies with each other. The Cilk work-stealing scheduler achieves execution space, time, and communication bounds all within a constant factor of optimal [6].

Cilk Plus was a convenient choice for several reasons. First, its task-oriented parallelism fits perfectly with our task-based recursion model. Second, Cilk Plus handles all task scheduling, so we must only spawn the parallel subproblems and let the built-in scheduler do the rest. Finally, Cilk Plus is supported by both ICC and GCC.

C. Framework Flow

The framework takes control of the algorithm's execution as soon as `solve()` is called. FRPA first checks if the base case should be executed: `runBaseCase()` should be called if `canRunBaseCase()` returns `true` and either `mustRunBaseCase()` returns `true` or there are no more characters remaining in the `interleaving` string. If the `interleaving` string is exhausted but `canRunBaseCase()` returns `false`, FRPA will continue to recurse with only DFS steps until `canRunBaseCase()` returns `true`.

The framework then checks whether to run the next level of the recursion in parallel (a BFS step) or sequentially (a DFS step). If a BFS step is required, FRPA calls `problem->split()` and spawns a new Cilk Plus task to solve the subproblems in each of the resulting `Tasks`. Finally, the framework calls `problem->merge()` to merge the subproblems.

If a DFS step is to be executed, the framework calls `problem->splitSequential()` and solves each of the resulting subproblems sequentially. Once solved, the framework calls `problem->mergeSequential()` to merge the subproblems. Recall that `mergeSequential()` and `splitSequential()` default to `merge()` and `split()` if they are not implemented by the `Problem`.

VI. Autotuning with OpenTuner

OpenTuner [7] is a new open source framework for building domain-specific multi-objective program autotuners. It supports fully-customizable configuration representations, an extensible technique representation to allow for domain-specific autotuning strategies, and an easy-to-use interface to communicate with the program to be tuned. A key capability of OpenTuner is its usage of a multitude of search techniques simultaneously; techniques that perform well will dynamically be allocated a larger proportion of tests than techniques that perform poorly. We use OpenTuner to tune FRPA algorithms offline, creating the optimal execution plan that can be queried from the OpenTuner SQL database at runtime.

A. General Usage

There are two important components to an OpenTuner autotuner. These components must be implemented in a small Python program that interfaces with the OpenTuner API. The first component is a `ConfigurationManipulator`, which defines the search space: OpenTuner will search over the parameter objects contained within the `ConfigurationManipulator`. The second component is the `run()` function, which is used to execute and evaluate the program with a specific configuration (specified by OpenTuner).

B. Tuning BFS/DFS Interleaving

FRPA is expected to be used in conjunction with OpenTuner to determine the optimal BFS/DFS parallelization strategy. The BFS/DFS interleaving is defined by two parameters on the OpenTuner `ConfigurationManipulator`: `schedule` and `depth`. `schedule` is a boolean array of length `MAX_DEPTH` that specifies the BFS/DFS interleaving. `depth` is an integer that specifies how deep the recursion should go (i.e. how many steps to take).

Listing 4 shows our implementation of the OpenTuner tuning script for Strassen-Winograd and CARMA matrix multiplication. The script accepts four parameters: the dimensions of the matrices (m, k, and n) and the maximum length of any BFS/DFS interleaving. The `run()` function reads the current configuration's BFS/DFS interleaving (provided by OpenTuner), runs the matrix multiplication algorithm with the specified interleaving, and returns $-1*$ GFlops (OpenTuner minimizes this return value, and minimizing $-1*$ GFlops results in maximizing performance). The `manipulator()` function constructs an OpenTuner `ConfigurationManipulator` that represents the search space. In this case, the `ConfigurationManipulator` contains the two parameters that define the interleaving: `schedule` and `depth`.

Note that there is an inherent inefficiency in the way we specify the BFS/DFS interleaving in an OpenTuner configuration. We wish to generate configurations that contain a boolean array of arbitrary length (i.e. an array of 'B's

```
import argparse
import opentuner
from opentuner.search.manipulator import *
from opentuner.measurement import
    MeasurementInterface

parser = argparse.ArgumentParser(parents=opentuner.
    argparsers())
parser.add_argument("--max_depth", type=int, default
    =10, help="max allowable depth of recursion")
parser.add_argument("--m", type=int, default=1024,
    help="m dimension of matrix")
parser.add_argument("--k", type=int, default=1024,
    help="k dimension of matrix")
parser.add_argument("--n", type=int, default=1024,
    help="n dimension of matrix")

class FRPATuner(MeasurementInterface):
  def __init__(self, args):
    super(FRPATuner, self).__init__(args)
    self.MAX_DEPTH = args.max_depth
    self.m = args.m
    self.k = args.k
    self.n = args.n

  def run(self, desired_result, input, limit):
    interleaving = ""
    cfg = desired_result.configuration.data
    for i in range(0, cfg["depth"]):
      if cfg["schedule"][i]:
        interleaving += "B"
      else:
        interleaving += "D"

    run_command = " ".join(["./harness", str(self.m)
        , str(self.k), str(self.n), interleaving])
    result = self.call_program(run_command)
    stdout = result["stdout"]
    gflops = float(stdout.split(",")[-1])
    return opentuner.resultsdb.models.Result(time
        =(-1*gflops))

  def manipulator(self):
    manipulator = ConfigurationManipulator()

    params = [
      IntegerParameter("depth", 1, self.MAX_DEPTH),
      BooleanArrayParameter("schedule", self.
          MAX_DEPTH),
    ]

    for param in params:
      manipulator.add_parameter(param)

    return manipulator

if __name__ == "__main__":
  args = parser.parse_args()
  FRPATuner.main(args)
```

Listing 4: tuner.py

and/or 'D's, where the i^{th} element of the array determines whether to take a BFS step or DFS step at depth i). However, at the time of writing, OpenTuner does not provide built-in, variable-length boolean array configuration parameters. Therefore, we choose to express a BFS/DFS interleaving with two parameters: `schedule` and `depth`. All instances of `schedule` are of length `MAX_DEPTH`, and the configuration parameter `depth` actually determines how deep the recursion is. For example, a configuration with `schedule` 'BBDB' and `depth` 3 represents the interleaving 'BBD'. This means that there exist distinct configurations that represent the same BFS/DFS interleaving. For instance, the configuration with `schedule` 'BBDB' and `depth` 3 represents the same interleaving as the configuration with `schedule` 'BBDD' and `depth` 3, namely 'BBD'. In our experiments, this redundancy did not seem to hinder OpenTuner's convergence on the optimal interleaving (see Section X).

C. Tuning Arbitrary Program Parameters

OpenTuner can also be used to autotune program-specific parameters in conjunction with the BFS/DFS interleaving. To add custom tuning parameters to the search, the `ConfigurationManipulator` must be initialized with the additional parameters and the `run()` function must use the user-defined parameters when executing the program.

VII. Case Study: Strassen-Winograd

The Strassen-Winograd algorithm [18] is a matrix multiplication algorithm that requires asymptotically fewer floating-point multiplications than classical matrix multiplication. The Strassen-Winograd version is usually preferred over the original Strassen algorithm because it requires fewer additions [1]. Each Strassen-Winograd multiplication problem splits into seven smaller matrix multiplication problems that can be solved recursively.

The algorithm is defined as follows (we are computing $C \leftarrow A * B$, where A is an $m \times k$ matrix and B is a $k \times n$ matrix). First partition A, B, and C into four quadrants:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \quad (1)$$

Then create the following temporary matrices:

$$\begin{aligned} T_0 &= A_{11} & S_0 &= B_{11} \\ T_1 &= A_{12} & S_1 &= B_{21} \\ T_2 &= A_{21} + A_{22} & S_2 &= B_{12} - B_{11} \\ T_3 &= T_2 - A_{11} & S_3 &= B_{22} - S_2 \\ T_4 &= A_{11} - A_{21} & S_4 &= B_{22} - B_{12} \\ T_5 &= A_{12} - T_3 & S_5 &= B_{22} \\ T_6 &= A_{22} & S_6 &= S_3 - B_{21} \end{aligned} \quad (2)$$

Then solve the following seven multiplications recursively:

$$\begin{aligned} Q_0 &= T_0 * S_0 & Q_3 &= T_3 * S_3 \\ Q_1 &= T_1 * S_1 & Q_4 &= T_4 * S_4 \\ Q_2 &= T_2 * S_2 & Q_5 &= T_5 * S_5 \\ & & Q_6 &= T_6 * S_6 \end{aligned} \quad (3)$$

Finally combine the results:

$$\begin{aligned} U_1 &= Q_0 + Q_3 & C_{11} &= Q_0 + Q_1 \\ U_2 &= U_1 + Q_4 & C_{12} &= U_3 + Q_5 \\ U_3 &= U_1 + Q_2 & C_{21} &= U_2 - Q_6 \\ & & C_{22} &= U_2 + Q_2 \end{aligned} \quad (4)$$

TABLE I: Machines used in this study.

Machine	Cores	Threads	CPU Type
Emerald	32	64	Intel Xeon X7560
Boxboro	40	80	Intel Xeon E7-4860

A. Implementation

Our implementation of Strassen-Winograd follows from the definition presented above[1]. An instance of `StrassenProblem` must store the following state: A, B, C, lda, ldb, ldc, m, k, and n. `split()` separates the matrices into quadrants specified in Equation 1, computes the temporary matrices shown in Equation 2, and initializes the seven subproblems defined in Equation 3. `merge()` performs the matrix operations specified in Equation 4 to solve the problem. The base case is single-threaded MKL.

In order to reduce the amount of memory required, care was taken to reuse already-allocated memory when possible. The temporary matrices T_0, T_1, T_6, S_0, S_1, S_5, Q_0, Q_2, Q_3, and Q_4 can all point to previously-allocated memory and need not be allocated at each recursive step.

B. Performance

Figure 3 shows Strassen-Winograd's performance for a selection of BFS/DFS interleavings on two machines (see Table I for hardware specifications). We multiply large, square, randomly-generated matrices with $m = k = n$ ranging from 1024 to 30 720 (in increments of 1024). We do not warm the cache before each trial, and the multiplication problems are sufficiently large so that it is not necessary to time batches of multiplications. We repeat each multiplication at least 5 times[2], and record the maximum performance across all trials (we wish to measure the algorithm's peak capability on a given machine).

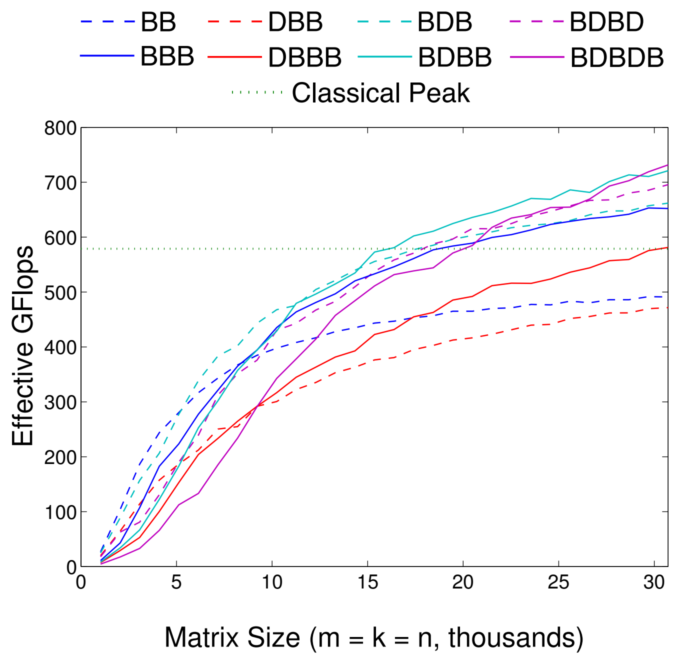
On the vertical axis, we plot *effective GFlops*. Effective GFlops represents the number of floating point operations per second (Flops) that would be achieved by a classical $O(n^3)$ matrix multiplication algorithm if it executed in the same amount of time. We compute effective GFlops according to the following formula[3]:

$$\text{effective GFlops} = \frac{m * k * n}{\text{time (seconds)} * 2^9} \quad (5)$$

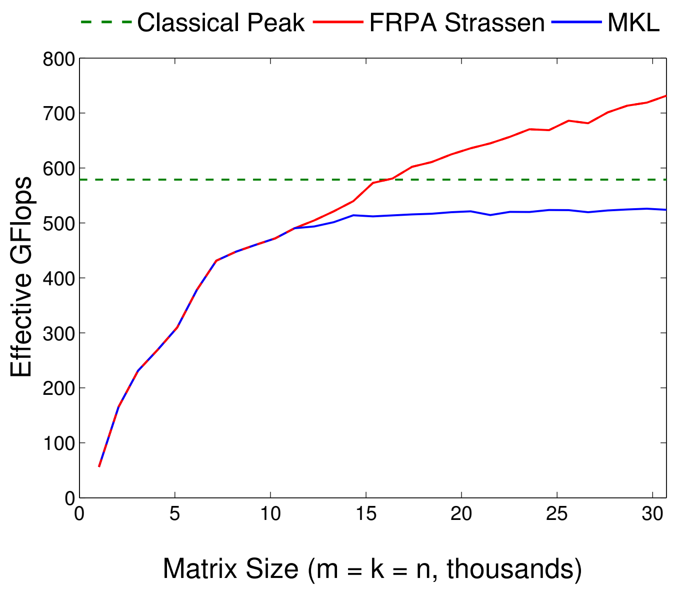
[1]For simplicity, our Strassen-Winograd implementation requires that matrix dimensions be divisible by 2^{depth}, where `depth` is the number of recursive steps (i.e. the length of the BFS/DFS interleaving). This constraint would be straightforward to remove.

[2]The variability of some matrix multiplications on our machines necessitated additional trials in order to accurately measure maximum performance. The average ratio of max performance to min performance over all trials was between 1.13 (double precision on Emerald) and 1.20 (single precision on Boxboro). The average standard deviation as a percent of mean (the coefficient of variation) ranged from 3.6% to 5.5%.

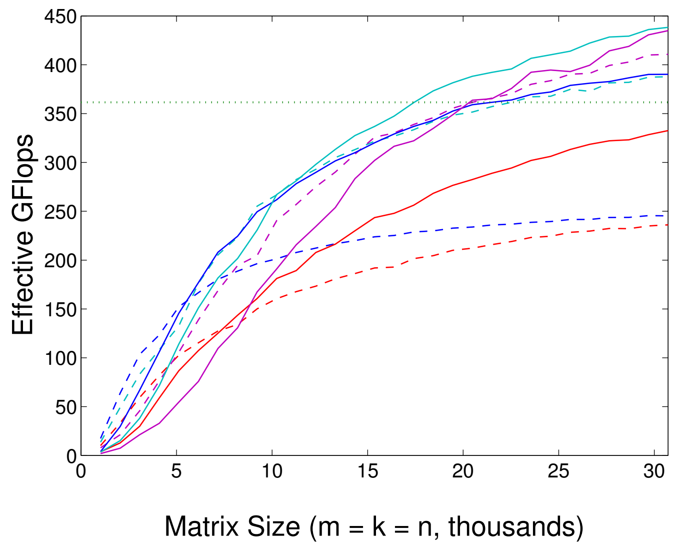
[3]For classical matrix multiplication, each entry of the $m \times n$ matrix C requires $2k$ floating point operations (k multiplies and k adds).



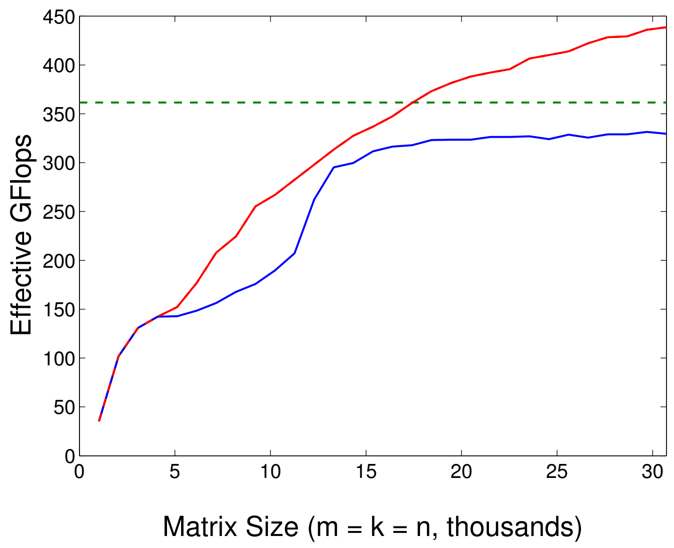
(a) Emerald, Single Precision



(a) Emerald, Single Precision



(b) Boxboro, Double Precision



(b) Boxboro, Double Precision

Fig. 3: Strassen-Winograd's performance for a selection of BFS/DFS interleavings on two machines. Effective GFlops is the number of floating point operations per second that a classical $O(n^3)$ matrix multiplication algorithm would perform if it completed the multiplication in the same amount of time. Classical peak is the maximum achievable performance of a classical algorithm. Dashed lines are interleavings that contain two BFS steps, and solid lines are interleavings that contain three BFS steps.

Fig. 4: Comparison of FRPA Strassen-Winograd with Intel's MKL. Strassen-Winograd outperforms MKL by up to 45% for large square matrices. Effective GFlops is the number of floating point operations per second that a classical $O(n^3)$ matrix multiplication algorithm would perform if it completed the multiplication in the same amount of time. For MKL, effective GFlops = actual GFlops because MKL employs a classical matrix multiplication algorithm. Classical peak is the maximum achievable performance of a classical algorithm.

Plotting effective GFlops allows us to compare Strassen-Winograd's performance with the theoretical maximum performance of any classical $O(n^3)$ algorithm (the horizontal line labeled "Classical Peak" in Figure 3). Classical peak is machine-specific.

In FRPA, OpenTuner selects the optimal BFS/DFS interleaving for each matrix size. This produces the solid red lines in Figure 4. For comparison, we also plot MKL's performance across the same set of matrix sizes. Our Strassen-Winograd implementation outperforms MKL by up to 45% for large square matrices (Boxboro, double precision).

Because the MKL routine employs an $O(n^3)$ matrix multiplication algorithm, it cannot exceed the classical peak (shown in dashed green). The Strassen-Winograd algorithm, however, executes asymptotically fewer floating point operations than any classical algorithm. As a result, the effective GFlops of our implementation of Strassen-Winograd exceeds the classical theoretical peak by up to 26% (Emerald, single precision).

Note that in Figure 4, MKL and Strassen-Winograd share a portion of the line (denoted by alternating red and blue dashes). This is because in FRPA, an algorithm executed with an empty BFS/DFS interleaving will immediately call the base case. For Strassen-Winograd, the base case the MKL matrix multiplication routine.

VIII. Case Study: CARMA

CARMA is a communication-optimal, cache-oblivious, parallel recursive rectangular matrix multiplication algorithm [3]. CARMA combines the largest-dimension-splitting technique of Frigo and Leiserson [19] with the recursive BFS/DFS approach introduced by Ballard, Demmel, Holtz, Lipshitz, and Schwartz [2].

CARMA is a simple recursive algorithm. At each recursive step, an $m \times k$ matrix A is multiplying a $k \times n$ matrix B to produce an $m \times n$ matrix C. The largest of these three dimensions is split in half, creating two subproblems of equal size that are solved recursively. The base case is solved with single-threaded MKL.

Figure 5 shows an example of each of the possible dimension splits. In each case, the largest dimension is divided to create two subproblems (the blue matrices represent one subproblem and the gold matrices represent the other).

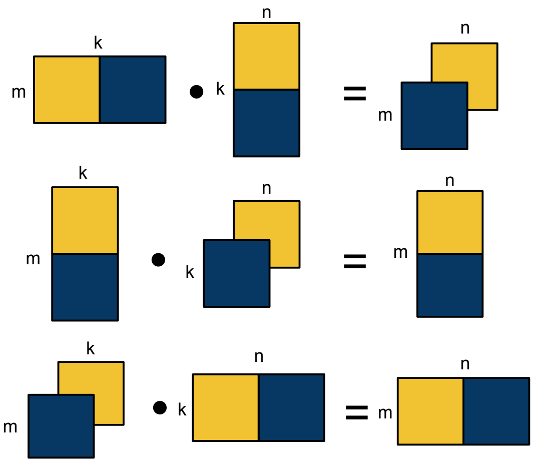


Fig. 5: Illustration of how a CARMA problem may split into subproblems. The first row demonstrates splitting in the k dimension, the second row shows splitting in the m dimension, and the third row represents splitting in the n dimension. In all cases, the longest of the three dimensions is split. The blue matrices represent one subproblem and the gold matrices represent the other

A. Implementation

CARMA's implementation in FRPA is straightforward[4]. An instance of `CarmaProblem` must store the following state: A, B, C, lda, ldb, ldc, m, k, and n. `split()` determines which of the three dimensions are largest, and splits the matrix multiplication into two subproblems accordingly. In the event that the k dimension is largest (the top example in Figure 5), a temporary matrix is required to hold the result of the second subproblem in order to prevent race conditions when executed in parallel. `merge()` must merge this temporary matrix with C if k was split. The base case is single-threaded MKL.

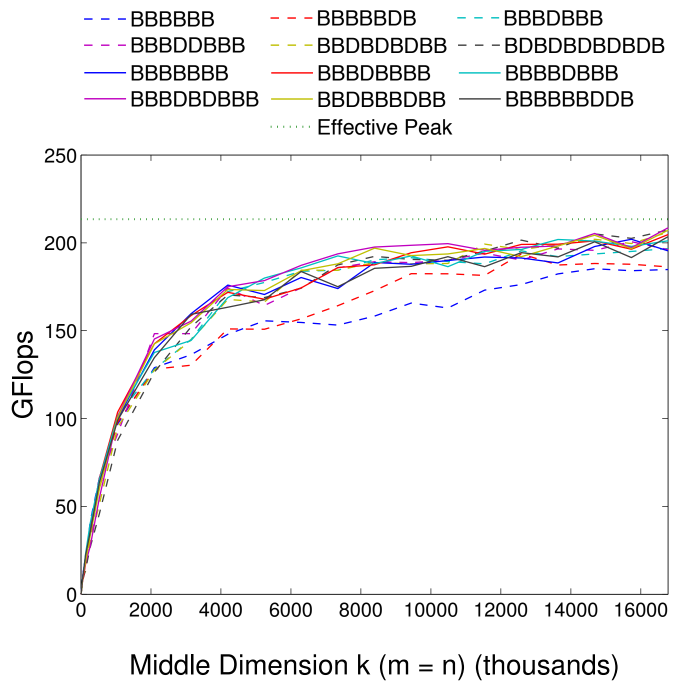
As described in Section III-B, we can perform an optimization if we know that the subproblems will not be executed in parallel (and therefore there can be no race conditions). We define `splitSequential()` to be the same as `split()`, with the exception of not creating the temporary matrix when the k dimension is split. Because there is no temporary matrix to merge and the multiplication happens in-place, `mergeSequential()` is a no-op.

B. Performance

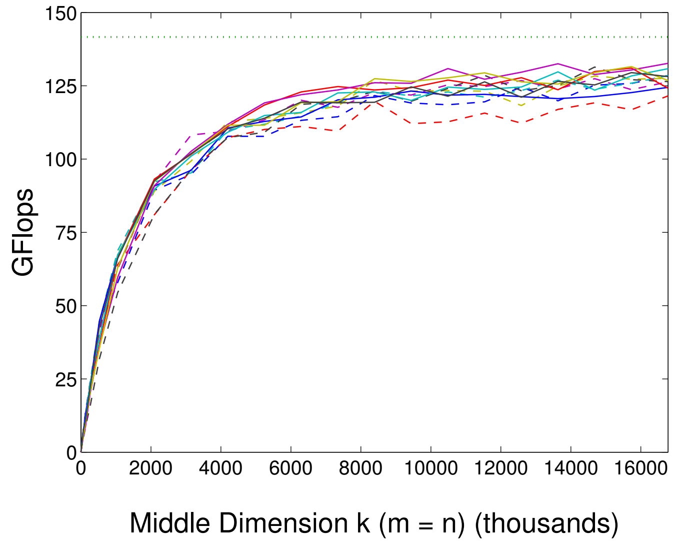
Figure 6 shows CARMA's performance for a selection of BFS/DFS interleavings on two machines (see Table I). We multiply randomly-generated "skinny" matrices, or matrices that have a large inner dimension k and small outer dimensions m and n (recall that multiplication is of the form $A * B$, where A is an $m \times k$ matrix and B is a $k \times n$ matrix). For this experiment, we set m and n to 64, and vary k up to 16 777 216 in increments of 1 048 576. We do not warm the cache before each trial. We repeat each multiplication at least 5 times[5], and record the maximum

[4]For simplicity, our CARMA implementation requires that matrix dimensions be divisible by 2^{depth}, where `depth` is the number of recursive steps (i.e. the length of the BFS/DFS interleaving). This constraint would be straightforward to remove.

[5]As with Strassen-Winograd, the variability of some matrix multiplications on our machines necessitated additional trials in order to accurately measure maximum performance. The average ratio of max performance to min performance over all trials was between 1.26 (single precision on Boxboro) and 1.34 (single precision on Emerald). The average standard deviation as a percent of mean (the coefficient of variation) ranged from 6.6% to 8.2%.

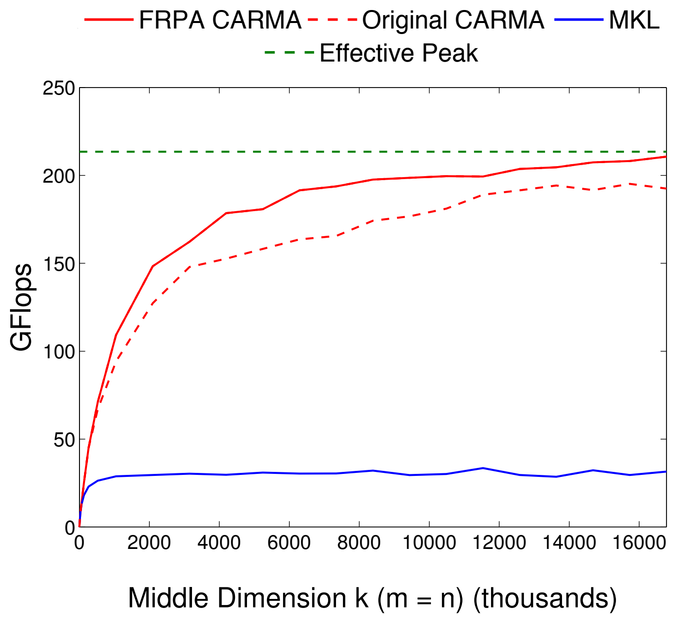


(a) Emerald, Single Precision

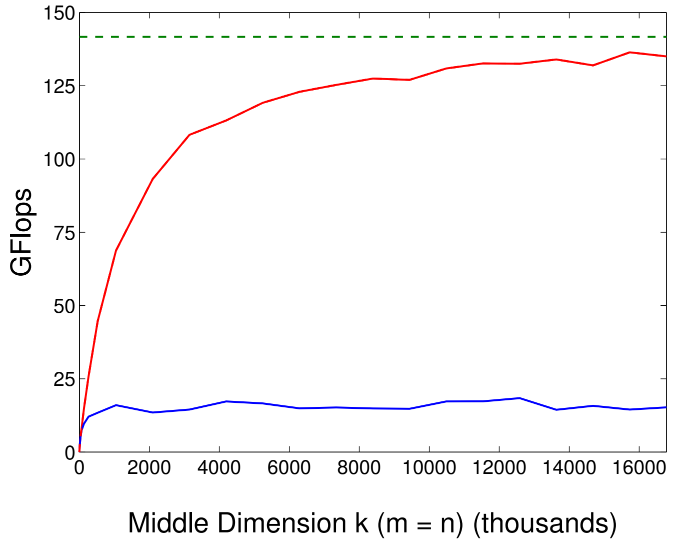


(b) Boxboro, Double Precision

Fig. 6: CARMA's performance on "skinny" matrices for a selection of BFS/DFS interleavings on two machines. Effective peak is the maximum achievable performance of CARMA on "skinny" matrices. Dashed lines are interleavings that contain six BFS steps, and solid lines are interleavings that contain seven BFS steps. Note that only the middle dimension, k, changes; m and n are set at 64.



(a) Emerald, Single Precision



(b) Boxboro, Double Precision

Fig. 7: Comparison of FRPA CARMA with original CARMA and Intel's MKL. Our new implementation outperforms the original by up to 56% and MKL by up to 11×. The original implementation of CARMA was only evaluated on Emerald, so we only compare the two implementations on Emerald. Note that only the middle dimension, k, changes; m and n are set at 64.

performance across all trials (we wish to measure the algorithm's peak capability on a given machine).

We plot GFlops (billion floating-point operations per second) on the vertical axis, which is computed by the following equation:

$$\text{GFlops} = \frac{m * k * n}{\text{time (seconds)} * 2^9} \tag{6}$$

We also show *effective peak*, which is the maximum achievable performance of a CARMA on "skinny" matrices. We compute effective peak on each machine by measuring single-threaded MKL for $m = k = n = 64$ and multiplying by the number of cores. This is an estimation of the best performance CARMA can achieve because the algorithm is limited by the base case multiplications, which all have $m = n = 64$.

As explained in Section VI, OpenTuner finds the optimal BFS/DFS interleaving for each matrix size. This produces the solid red lines in Figure 7. For comparison, we also plot the performance of the original CARMA implementation (described in [3]). FRPA's implementation surpasses the original algorithm (by up to 56%) because the original version only executes BFS steps whereas FRPA tunes the parallelization strategy.

We also show MKL's performance in Figure 7. CARMA outperforms MKL by up to 11× (Boxboro, single precision) due to the communication-optimal BFS/DFS approach [3]. MKL and CARMA share a portion of the line (represented by alternating red and blue dashes). This is because the MKL matrix multiplication routine is the base case in CARMA.

IX. Analysis of Performance Variation

As we expected, performance varied across different BFS/DFS interleavings. We attribute this variation to four factors, described in this section: the number of Cilk tasks spawned, the size of base case problems, the memory footprint, and the cache access pattern.

A. Number of Cilk Tasks

One source of variability is the number of Cilk tasks spawned. As more tasks are spawned, Cilk's load balancer has more flexibility and fine-grained control. However, excessive task spawning introduces overhead that can outweigh the benefits. Each additional BFS step increases the number of Cilk tasks exponentially. In algorithms with high branch factors like Strassen-Winograd (which spawns 7 subtasks), this can hurt performance.

For example, compare an interleaving with ten BFS steps to one with only five. In Strassen-Winograd, the interleaving with five BFS steps spawns 7^5 (approximately 17 thousand) Cilk tasks; the interleaving with ten BFS steps spawns 7^{10} tasks (over 280 million). Seventeen thousand tasks is clearly enough to fully utilize the processors, and the overhead of spawning 280 million tasks is insurmountable.

These effects can be measured by comparing the performance of two interleavings of the same length, but with different numbers of BFS steps. For example, consider the two interleavings 'BDBDBDBDBD' (five 'B's) and 'BBBBBBBBBB' (ten 'B's), each of which recurses to a depth of ten. The former outperforms the latter: for double-precision Strassen-Winograd on Boxboro at $m = k = n = 10\,240$, for example, the interleaving with five BFS steps is 42% faster.

B. Base Case Size

Another factor that affects performance is the size of the problem when it reaches the base case. Depending on the base case routine and the problem itself, certain sizes of inputs to the base case may perform better than others. As the recursion gets deeper (regardless of which steps are used), the problems at the base case become smaller. This has a slightly negative effect on performance in both CARMA and Strassen-Winograd: all else equal, the single-threaded MKL routine is more efficient operating on larger matrices.

C. Memory Allocation

Memory allocation also affects overall algorithm performance: all else equal, more memory allocation increases overhead. This effect is difficult to isolate; in most cases, "all else" is not equal because changing the BFS/DFS interleaving also affects the other three factors (number of Cilk tasks, base case size, and cache access patterns). For example, executing a BFS step rather than a DFS step in CARMA increases total memory usage, but enables additional parallelism and may be necessary to generate enough Cilk tasks for the algorithm to efficiently use the available processors.

The effect of memory allocation on performance can be somewhat isolated when comparing BFS/DFS interleavings that differ in recursive depth. For example, the Strassen-Winograd interleavings 'BBB' and 'DBBB' differ by only the DFS step at the beginning. This DFS step causes a significant increase in memory allocation (nearly a factor of two), yet does not expose additional parallelism or opportunity for load balancing. As a result, 'BBB' outperforms 'DBBB' for all matrix sizes in our tests.

D. Cache Access Patterns

Finally, the cache access pattern (which represents communication in shared-memory architectures), also affects performance. Changing the BFS/DFS interleaving and recursion depth significantly affects data locality, and therefore cache performance.

Although it is difficult to isolate this factor from the others, the impact of cache locality is apparent in some instances. For example, consider the BFS/DFS interleavings 'DBDB' and 'BDBD' for Strassen-Winograd. These interleavings spawn the same number of Cilk tasks, solve base cases of the same size, and allocate the same amount of memory. However, at $m = k = n = 10\,240$ on Emerald (double precision), the 'BDBD' interleaving outperforms the 'DBDB' interleaving by 31%. Careful analysis shows that the 'BDBD' execution results in 51% fewer L3 cache misses than the 'DBDB' interleaving. This explains the performance difference we observe.

We measure L3 cache miss rates using the Performance Application Programming Interface (PAPI) [20], version 84017152. In order to avoid the effects of prefetching on PAPI's counters, we disable hardware prefetching on our machine via the BIOS. PAPI only measures cache misses from a single thread, but because our algorithm spawns many tasks and because Cilk Plus employs a work-stealing scheduler, one thread's L3 cache misses is roughly proportional to total L3 cache misses. We repeat each multiplication ten times and average the results (we observed a standard deviation of 6% to 8% of mean).

To validate that we are properly measuring L3 cache misses, we ran a simple experiment. We created a program that allocates two arrays of equal size and and a stride length of one, fills them both with random floating-point numbers between -1 and 1 (to warm the cache), and repeatedly copies one array into the other[6]. We then instrumented the array copy using our approach (i.e. disabling hardware prefetching and reporting PAPI's L3 total cache miss counter, `PAPI_L3_TCM`).

We vary the array sizes from 32KB to 50MB each and measure the number of cache misses that occur while copying one array into the other (after warming the cache). We repeat each array copy 100 times and report the average number of cache misses. As expected, once the sum of the array sizes grows somewhat larger than the L3 cache size (24MB on Emerald), the number of measured cache misses approaches $2 * \frac{\text{Array Size}}{64}$ (Emerald's L3 cache line is 64 bytes long). This indicates that our technique for measuring L3 cache misses is valid. See Figure 8 for an illustration.

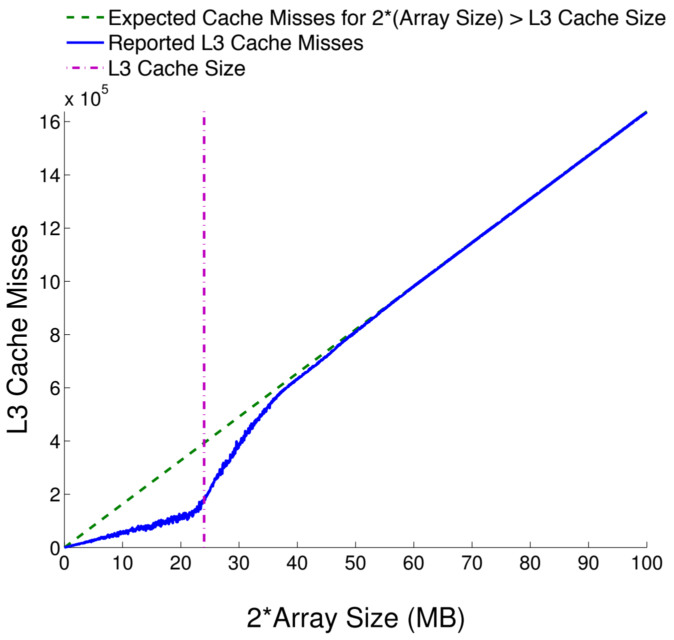


Fig. 8: Validation of our L3 cache miss measuring technique. We allocate two arrays of size ranging from 32KB to 50MB each, warm the cache, and measure the number of cache misses that occur while copying one array into the other. We repeat the array copy 100 times and plot the average number of cache misses. As expected, once the sum of the array sizes grows somewhat larger than the L3 cache size (24MB on Emerald), the number of measured cache misses approaches $2 * \frac{\text{Array Size}}{64}$ (Emerald's L3 cache line is 64 bytes long). For $(2 * \text{Array Size}) < (\text{L3 Cache Size})$, we observe very few cache misses (as expected) for roughly two-thirds of the repetitions, and $2*\frac{\text{Array Size}}{64}$ cache misses for the other one-third (for unknown reasons), resulting in the line shown in the figure when the iterations are averaged.

X. OpenTuner Convergence

Figure 9 shows OpenTuner's convergence at a single datapoint (Strassen-Winograd, $m = k = n = 4096$, single precision, Emerald). The solid line represents the median (over 100 trials[7]) of the fastest BFS/DFS interleaving discovered after tuning iteration i, for i ranging from 1 to 100. Each tuning iteration represents OpenTuner trying a single configuration (i.e. BFS/DFS interleaving) for the specified matrix dimensions[8], and we plot the performance of the best configuration found so far (performance can only increase with additional iterations). The error bars mark the first and third quartiles at each tuning iteration, and the maximum performance is shown as a dashed green line. OpenTuner's `MAX_DEPTH` (the longest allowable BFS/DFS interleaving) is 10, and the performance of each configuration is recorded as the maximum of three multiplications. At this datapoint, which is roughly representative of most, OpenTuner converges to 95% of optimal after 18 iterations.

XI. Future Work

We have identified multiple areas for potential future work including further analyzing FRPA's capabilities, implementing FRPA in existing frameworks, expanding FRPA's feature set, and extending FRPA to other architectures.

A. Analyze Performance of Additional Algorithms

We only performed detailed performance analyses on two algorithms (Strassen-Winograd and CARMA). We hypothesize that the performance of the other six recursive algorithms we implemented are comparable to other well-

[6]This procedure was inspired by the STREAM Benchmark [21].

[7]Because the tuning process is nondeterministic, we repeated it 100 times for this experiment.

[8]OpenTuner must tune each value of (m, k, n).

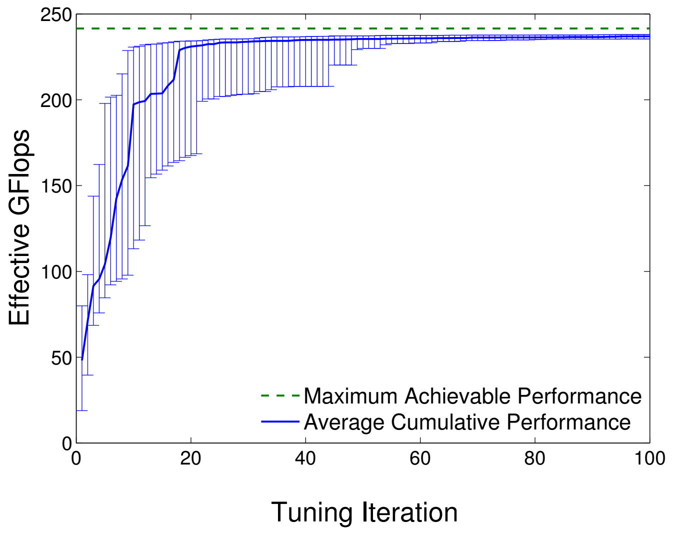


Fig. 9: OpenTuner convergence (Emerald, single precision Strassen-Winograd, $m = k = n = 4096$). The solid line represents the median (over 100 trials) of the fastest BFS/DFS interleaving discovered after tuning iteration i, for i ranging from 1 to 100. The error bars mark the first and third quartiles at each tuning iteration. OpenTuner converges to 95% of optimal after 18 tuning iterations.

tuned versions of the algorithms, and we leave this analysis for future work.

B. *Explore Benefits of Parallelizing `split()` and `merge()`*

We only explored the effects of paralleling `split()` and `merge()` in our implementation of Strassen-Winograd. Due to the fact that the algorithm's runtime is dominated by the base case matrix multiplications, this optimization had no measurable performance impact. It would be interesting to find an algorithm in which performance is bottlenecked by the `split()` and `merge()` functions, and then evaluate the effect of parallelizing `split()` and `merge()`.

C. *SEJITS Integration*

Work is in progress to create a SEJITS [22] specializer to extend the framework's API to be accessible from Python. Domain scientists hoping to write parallel recursive algorithms would implement the `split()`, `merge()`, and `runBaseCase()` functions in Python. These input functions would be used to generate efficient C++ code.

D. *Nonuniform BFS/DFS Interleavings*

Currently FRPA executes all subproblems at a given level using the same parallelization strategy (i.e. breadth-first or depth-first). A more complicated parallelization strategy that executes some subproblems at the same depth in parallel and others sequentially is possible. This may be especially beneficial in problems with non-homogeneous subproblems. In TRSM, for example, it may be favorable to execute the `TrsmProblem` subproblems in parallel and the `MultProblem` subproblems sequentially. We leave it to future work to explore these tradeoffs.

E. *Distributed Memory Implementation*

We hope to explore a similar framework for parallel recursive algorithms on distributed-memory systems. Spark [23], an open-source data analytics cluster computing framework, may provide an appropriate platform for FRPA. This distributed-memory framework would be logically equivalent to our current one, but would be able to support larger computations and datasets on distributed machines.

F. *Heterogeneous Architectures*

Many algorithms can achieve very high performance on heterogeneous architectures comprising multi-core CPUs and GPUs. We believe adding support for heterogeneous architectures could make FRPA an even more powerful tool for generating high-performance code.

XII. Conclusion

FRPA combines Cilk Plus, OpenTuner, and communication-avoiding parallelization techniques to provide a powerful API for writing recursive algorithms. We validate the framework's generality by implementing eight algorithms: Strassen-Winograd, CARMA, mergesort, quicksort, TRSM, SYRK, Cholesky decomposition, and Delaunay triangulation. An in-depth study of the performance characteristics of our Strassen-Winograd and CARMA implementations demonstrates FRPA's viability as a tool for developing high-performance algorithms. Both implementations outperform the Intel MKL dense matrix multiplication code (CARMA by up to 11× and Strassen-Winograd by up to 45%). Moreover, the FRPA implementation of CARMA outperforms the original CARMA implementation (by up to 57%) due to the ability to tune the BFS/DFS interleaving.

The Delaunay triangulation implementation was written by a researcher who did not contribute to the development of the framework itself. Although this is not an exhaustive analysis, it does reveal that FRPA can be useful without knowledge of its internal machinery.

We envision FRPA as a component of a larger Selective, Embedded Just-in-Time Specialization (SEJITS) framework, namely A SEJITS implementation for Python (Asp). At its core, FRPA is about making high-performance computing accessible to individuals who may have limited computer science backgrounds. Asp takes that a step further by providing a mechanism to generate efficiency-language code from Python source code [22], [24]. FRPA and SEJITS act as the bridge between domain experts and high-performance computing.

XIII. Acknowledgments

Michael Driscoll helped with the OpenTuner integration and general guidance. Andrew Gearhart, Kostadin Ilov, and Dr. Shoaib Kamil assisted in finding a way to accurately measure L3 cache misses on Emerald. Ben Lipshitz provided early guidance and the implementation of the Delaunay Triangulation algorithm. Dr. Shoaib Kamil and Dr. Oded Schwartz provided direction at the beginning of the project. Peter Birsinger assisted in the development of the framework and wrote an initial version of the Strassen-Winograd matrix multiplication algorithm. Kevin Sheu implemented the initial version of the memory tracking feature. At the time of writing, David Dinh is in the process of implementing FRPA as a SEJITS [22] specializer.

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research (a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA), and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Nokia, NVIDIA, and Oracle. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

Appendix: Algorithm Details

We implemented a total of eight algorithms using FRPA. Strassen-Winograd and CARMA were discussed in detail in Section VII and Section VIII, respectively. In this section, we provide an overview of the other six algorithms and how we implemented them.

A. Mergesort

Mergesort is a classic sorting algorithm. To sort an array, mergesort splits the array in half into two subproblems, recursively sorts each subproblem, and merges the two results into the final sorted array.

An instance of `MergesortProblem` contains the following state: A (the array to be sorted) and *length* (the length of that array). `split()` splits the array in half, `runBaseCase()` calls the single-threaded `qsort()` function, and `merge()` merges the two sorted subproblems.

Our implementation of mergesort is 60 lines of code and has been verified to correctly sort arrays of arbitrary length. See Listings 1, 2, and 3 for our full implementation of mergesort.

B. Quicksort

Quicksort is another classic sorting algorithm. In quicksort, the array is partitioned into two arrays of roughly equal size, where every value in one array is less than every value in the other array. Subproblems are solved recursively until either the recursion has reached its maximum depth or the array to be sorted has fewer than two elements.

An instance of `QuicksortProblem` contains the following state: A (the array to be sorted) and *length* (the length of that array). `split()` randomly selects a pivot and partitions the array into one array with values less than the pivot and one array with values greater than the pivot. `runBaseCase()` calls the single-threaded `qsort()` function, and `merge()` is a no-op (the array is sorted in-place). Our implementation of quicksort is 45 lines of code and has been verified to correctly sort arrays of arbitrary length.

C. TRSM

TRSM (triangular matrix system solver) performs a triangular matrix solve: given an input matrix X_{in} and lower non unit triangular matrix T, our implementation of TRSM solves the equation $X_{out} * T' = X_{in}$. X_{out} replaces X_{in} in memory.

The algorithm is defined as follows. First partition X and T into four quadrants:

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \qquad T = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} \tag{7}$$

Note that $T_{12} = 0$ and that T_{11} and T_{22} are lower triangular. TRSM may then be defined recursively:

```
trsm(X, T):
    trsm(X11, T11)
    mult(X12, X11, T21)
    trsm(X12, T22)

    trsm(X21, T11)
    mult(X22, X21, T21)
    trsm(X22, T22)

mult(C, A, B):
    C = C - A * B'
```

The first set of three subproblems and the second set of three subproblems are independent of each other and may run in parallel. However, the three subproblems within each set must be executed sequentially. Therefore, each set of three subproblems represents a `Task` in our framework (see Section IV-A for a discussion of `Tasks` vs `Problems`).

An instance of `TrsmProblem` contains the following state: X, ldx, T, ldt, and n (the size of the matrices). `split()` partitions the matrices as shown in Equation 7 and initializes two `Tasks` with three subproblems each. `runBaseCase()` calls Intel MKL's `trsm()` function, and `merge()` is a no-op (the computation is performed in-place).

TRSM is an example of algorithm composition because subproblems of `TrsmProblem` are of type `TrsmProblem` and `MultProblem`. `MultProblem` is defined independently, and executes MKL's `gemm()` to perform the multiplication.

Our recursive implementation of TRSM is only 40 lines of code (not including 20 lines for MultProblem) and has been verified as correct.

D. SYRK

Our slightly simplified version of symmetric rank-k matrix-matrix operation (SYRK) takes input matrix A

and lower triangular matrix C and performs the operation $C = C - A * A^T$. The A and C matrices are partitioned into quadrants much like the matrices in TRSM:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \tag{8}$$

SYRK may now be defined recursively:

```
syrk(C, A):
    syrk(C11, A11)
    syrk(C11, A12)
    mult(C21, A21, A11)
    mult(C21, A22, A12)
    syrk(C22, A21)
    syrk(C22, A22)

mult(C, A, B):
    C = C - A * B'
```

All six subproblems may be executed in parallel provided a temporary matrix is used for the duplicated outputs (C_{11}, C_{21}, and C_{22}).

An instance of `SyrkProblem` contains the following state: A, lda, C, ldc, and n (the size of the matrices). `split()` partitions the matrices as shown in Equation 8 (allocating temporary matrices for the duplicate C outputs) and initializes six `Task`s with one subproblems each. `merge()` must combine the temporary output matrices and free them. `runBaseCase()` calls Intel MKL's `syrk()` function.

Like CARMA, SYRK can take advantage of the `splitSequential()` and `mergeSequential()` optimization (see Section III-B). If the subproblems will be executed sequentially (i.e. during a DFS step), we need not allocate temporary matrices to hold the duplicated output. Therefore, we implement `splitSequential()` like `split()` but without the extra allocation, and we define `mergeSequential()` to be a no-op.

Like TRSM, SYRK is an example of algorithm composition: subproblems of `SyrkProblem` are of type `SyrkProblem` and `MultProblem`. Our recursive implementation of SYRK is 90 lines of code (not including 20 lines for MultProblem) and has been verified as correct.

E. Cholesky

Cholesky decomposition is the decomposition of a symmetric positive-definite matrix A into the product of a lower triangular matrix L and its transpose (i.e. $A = L*L^T$). The A matrix is partitioned into quadrants exactly as in SYRK:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{9}$$

We now define Cholesky recursively:

```
cholesky(A):
    cholesky(A11)
    trsm(A21, A11)
    syrk(A22, A21)
    cholesky(A22)
```

All subproblems of `CholeskyProblem` must be solved sequentially. Therefore, `split()` partitions the A matrix as defined in Equation 9 and creates a single task containing all four subproblems. The parallelism is exposed further into the recursion when the TRSM and SYRK subproblems are solved. `runBaseCase()` calls MKL's `potrf()` function, and `merge()` is a no-op (the calculation happens in-place). An instance of `CholeskyProblem` contains the following state: A, lda, and n (the size of the matrix).

Cholesky decomposition is yet another example of algorithm composition. The ability to reuse our previously-defined `TrsmProblem` and `SyrkProblem` (which both rely on algorithm composition themselves) demonstrates the ease with which developers can compose problems in FRPA. Due to our ability to abstract away the other types of recursive problems that `CholeskyProblem` uses, our implementation of Cholesky decomposition is only 30 lines of code[9]. Our implementation has been verified as correct.

F. Delaunay Triangulation

Delaunay Triangulation is a triangulation method that maximizes the angles of the triangles (i.e. it avoids creating long, skinny triangles if possible) [5]. Benjamin Lipshitz implemented this algorithm using FRPA: `split()` divides the points spatially into left and right halves, `merge()` merges the two sets of triangles together from bottom to top, and `runBaseCase()` draws the line if there are two points, or draws the triangle if there are three points. Because `runBaseCase()` may be called only if there are three or fewer points, we overrode `canRunBaseCase()` in our implementation. Our implementation is 580 lines of code and has been verified as correct.

References

[1] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for Strassen's matrix multiplication," in *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. New York, NY, USA: ACM, 2012, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/2312005.2312044

[2] G. Ballard, J. Demmel, B. Lipshitz, and O. Schwartz, "Communication-avoiding parallel Strassen: Implementation and performance," in *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 101:1–101:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389133

[3] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication," in *IEEE International Parallel & Distributed Processing Symposium*, 2013.

[4] Intel, "Math Kernel Library," http://software.intel.com/en-us/intel-mkl.

[5] L. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi," *ACM Transactions on Graphics (TOG)*, vol. 4, no. 2, pp. 74–123, 1985.

[9]The other types of problems that `CholeskyProblem` uses directly (`TrsmProblem` and `SyrkProblem`) or indirectly (`MultProblem`) sum to 150 lines of code.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system.* ACM, 1995, vol. 30, no. 8.

[7] J. Ansel, S. Kamil, K. Veeramachaneni, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," 2013.

[8] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, 1996.

[9] G. Bilardi, A. Pietracaprina, G. Pucci, and F. Silvestri, "Network-oblivious algorithms," in *Proceedings of 21st International Parallel and Distributed Processing Symposium*, 2007.

[10] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran, "Oblivious algorithms for multicores and network of processors," in *IPDPS*, 2010, pp. 1–12.

[11] R. Cole and V. Ramachandran, "Resource oblivious sorting on multicores," in *Proceedings of the 37th international colloquium conference on Automata, languages and programming*, ser. ICALP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 226–237. [Online]. Available: http://dl.acm.org/citation.cfm?id=1880918.1880944

[12] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *SC 2006 Conference, Proceedings of the ACM/IEEE.* IEEE, 2006, pp. 5–5.

[13] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE International Conference on.* IEEE, 2008, pp. 142–151.

[14] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International.* IEEE, 2011, pp. 676–687.

[15] T. Gautier, J. V. F. Lima, N. Maillard, B. Raffin *et al.*, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[16] R. L. Collins, B. Vellore, and L. P. Carloni, "Recursion-driven parallel code generation for multi-core platforms," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010.* IEEE, 2010, pp. 190–195.

[17] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *PetaBricks: a language and compiler for algorithmic choice.* ACM, 2009, vol. 44, no. 6.

[18] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of symbolic computation*, vol. 9, no. 3, pp. 251–280, 1990.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on.* IEEE, 1999, pp. 285–297.

[20] B. D. Garner, S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, pp. 189–204, 2000.

[21] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[22] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "SEJITS: Getting productivity and performance with selective embedded jit specialization," *Programming Models for Emerging Architectures*, 2009.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[24] S. Kamil, D. Coetzee, and A. Fox, "Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization," in *Python for Scientific Computing Conference (SciPy)*, 2011.